Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

A general-purpose modifiable AST parser

Michael Doberstein

Bachelorarbeit

Beginn der Arbeit:	29th of September 2024
Abgabe der Arbeit:	30th of December 2024
Gutachter:	Dr. John Witulski
	Dr. Jens Bendisposto

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30th of December 2024

Michael Doberstein

Abstract

Software developers are often facing the issue of maintaining large amount of source code. The keep the source code up to date, often times a process called refactoring is needed, in which the software developers update the code.

To allow the software developers to perform these updates automatically, most integrated development environments (IDE) provide tools to perform predefined refactorings automatically. The effort of performing some refactorings on large code bases can be too much to perform manually. Most of the times, the tools provided by the IDE are sufficient to perform the refactorings, but sometimes a custom solution is required. One way to create automated custom refactorings is by parsing the source code and building it into an abstract syntax tree (AST), which can be manipulated.

Here we provide a Java library, which can parse a subset of the context-free language based on their corresponding grammar definition and generates a AST which can be modified by the user and afterwards reverted back to the source code.

The program provides a relatively easy method of safe large scale refactorings, however the effort of defining a grammar is large. It also has been shown, that this tool can also be used to extend a given programming language and allow for additional features to be added, basicly allowing for a primitive form of source to source compilation.

Erklärung zur Nutzung generativer KI

Im Rahmen der vorliegenden Bachelorarbeit wurde generative KI zu folgenden Zwecken genutzt:

- DeepL Write zur sprachlichen Überarbeitung der Arbeit
- Machine learning assistant von der IntelliJ IDE wurde bei Implementierung des Programmes verwendet

Acknowledgements

Im Falle, dass Sie Ihrer Arbeit eine Danksagung für Ihre Unterstützer (Familie, Freunde, Betreuer) hinzufügen möchten, können Sie diese hier platzieren.

Dieser Part ist optional und kann im Quelltext auskommentiert werden.

Contents

1 Introduction

For this thesis a Java library called `gp-modifiable-ast` was implemented. `gp` stands for general purpose, meaning that it can provide the API for different grammars, e.g. for different programming languages.

1.1 Motivation

The motivation for this work came from a practical problem. During the development of a large enterprise web application, there was a need to change the translation system. A translation system receives a token as a string and returns a predefined translation depending on the target language. This caused some problems because the new translation system had a different call syntax than the old one. This refactoring required looking at external data stored in a database for each different call to the translation function. Because of the dependency on external data, the automatic tools provided by the IDE were not sufficient to refactor these calls.

On the other hand, this would have been a very time-consuming task to do manually, as there were over ten thousand calls to this function.

To solve this problem, a special JavaScript parser like Esprima [esp] was used. Esprima can parse JavaScript, provide an AST, allow modifications on the AST, and convert back to source code. With this library, it was possible to perform the refactoring with the necessary customization options and accuracy.

However, tools like Esprima exist mostly for popular programming languages, and each tool provides a different API.

While there are parser generators like SableCC [sab] that can generate parsers from a definition file, they are intended to be used for a different purpose. Common parser generators are implemented to provide part of a compiler front end, but they do not need to maintain whitespace, comments, and be able to generate the original source code from the AST.

The goal of this work is to provide a library called `gp-modifiable-ast` to minimize the effort needed to be able to perform manipulations directly on the AST and to be able to transform the AST back into source code while maintaining all whitespaces and comments.

The following example is intended to illustrate that rewritable abstract syntax trees can be used for refactorings that require some knowledge of the program structure. The change that should be made to the following source code is to rename all variables to enforce a camel case naming scheme for variables, while keeping Pascal case for function names. While this can be done with IDE tools, it would require manual effort each time.

Implementing a custom solution can enforce this naming scheme automatically.

```
void should_be_snake_case(int should_be_camel_case) {
    int second_variable = should_be_camel_case;
}
```

Listing 1: Example of untransformed code

The resulting AST from this code might look like this:

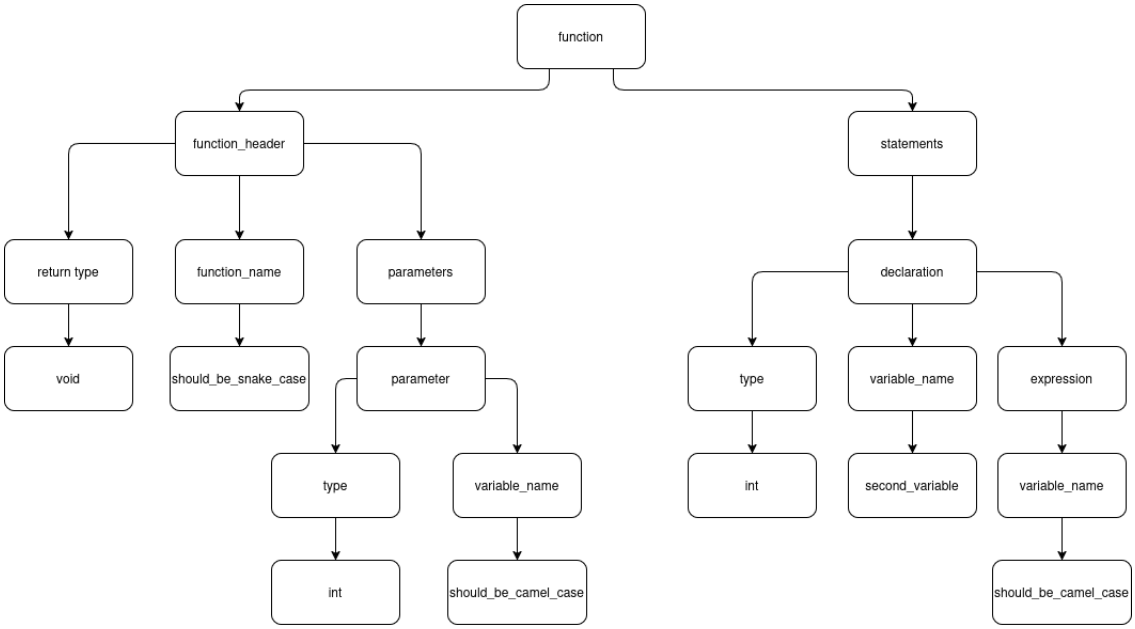

Figure 1: AST example

This AST contains a lot of information about the program structure and allows you to distinguish between variable and function names. [1].

The following code should give a general idea of how transformations could be implemented on the AST. The API of `gp-modifiable-ast` is different from this example.

```
    AST ast = parser.parse("example.java");
    List<ASTNode> nodes = ast.find("variable_name");
    renameToCamelCase(nodes);
    ast.transformBackToSourceCode();
```

Listing 2: Example of transformation

This transformation would result in the following code.

[1] Depending on the grammar and language, this differentiation may not always be possible on the generated AST. For example in JavaScript, functions can be passed as parameters into other functions: `function_a(function_b, variable_a)`. The implemented LR(1) parser is not able to differentiate between the types of the parameters in this case.

```
void should_be_snake_case(int shouldBeCamelCase) {
    int secondParam = firstParam;
}
```

Listing 3: Example of transformation

This example should illustrate why rewritable abstract syntax trees can be a useful tool for software development, because they allow a wide range of refactorings by providing access to different pieces of information about the program structure.

1.2 Related work

The library implemented for this thesis (`gp-modifiable-ast`) has similarities to common parser generators such as SableCC [sab], ANTLR [ant], GNU Bison [gnu] and others.

There are custom parsers like Esprima [esp] that can generate a modifiable AST for JavaScript sources, but these are limited to a specific programming language.

`gp-modifiable-ast` is able to preserve the original formatting as much as possible and can be defined for any grammar that can be parsed by an LR(1) parser.

This work is based on the conference paper by Jeffrey L. Overbey and Ralph E. Johnson with the title "Generating Rewritable Abstract Syntax Trees" [OJ09]. This paper defines a possible grammar definition for rewritable abstract syntax trees, which is partially implemented by `gp-modifiable-ast`. Some of the proposed definitions of [OJ09] are not implemented, as they would not make sense with the structure of `gp-modifiable-ast`. The chapter 3.5 discusses which are implemented and which are not.

2 Prerequisites

In this chapter we want to establish the prerequisites required to understanding this thesis.

2.1 Regular Expressions and Grammars

Based on the paper of Naom Chomsky [Cho56] a grammar G consists of four components.

1. A finite set N of nonterminal symbols.
2. A finite set Σ of terminal symbols.

3. A finite set P of production rules, each defined as $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, where * is the kleene closure operator [App02].
4. A start symbol $S \in N$.

By limiting the production rules P, several classes can be defined based on the chomsky hierarchy [Lin02]. The main classes we are using in this thesis is the set of regular languages and the set of context free grammars.

Regular languages are languages that can be described by a grammar where every $p \in P$ is defined as $N \rightarrow (Ba)$, where $B \in \Sigma$ and $a \in N$.

In theoretical computer science, a grammar consists out of a set Σ of terminal symbols, a set N of nonterminal symbols, a set P of productions and a start symbol S. S is always a nonterminal symbol and therefore $S \in N$. The productions are defined as $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$. The $^+$ refers to one or more, the * refers to zero or more.

Grammars can be seperated in different different classes based on the chromsky hierarchy. The main ones we are looking at in this thesis are the contextfree grammars and the regular grammars.

A grammar is context free, if for all productions $p \rightarrow q$ in P the following applies: $p \in N$.

A grammar is a regular grammar, if for all prodution $p \rightarrow q$ in p the following applies: $p \in N$ and $q \in \Sigma \cup \Sigma N$.

Grammars are used, because they can describe a language. In our use case, they describe an programming language and can be used, to parse the source code. Regular expressions are short statements, that avoid defining an entire grammar definition. Their downside is, that they are not able to describe alot of the features modern programming languages have. For example, it is not possible to write a regular grammar for the language that has the same amount of opening and closing brackets. An example for a regular expression would be this:

$$a^*ba^*b$$

This would parse all words that have any amount of "a", followed by exactly one b, followed by any amount of "a", followed by exactly one b.

To handle these situations, we require the context free grammars, they are able to parse more cases and are sufficient enough, to parse most programming languages. However, even the context free grammars are not powerfull enough to correctly verify if a given source code is valid. For this, we would technically need context sensitive grammars. Those come with the cost of a worse runtime, therefore usually context free grammars

are used when writing compilers and additional verifications are done after the parsing to ensure the language is indeed valid source code.

An example for a context free grammar:

Let $\Sigma = \{E\}, N = \{a, b\}, S = \{E\}$ and the productions defined as:

$$P = \{E \to aEb, \\ E \to \epsilon\}$$

ϵ is a special symbol, it refers to the empty word. That means, that E can be derived to nothing. Without this, we would have endless self recursion in the grammar rule.

This grammar would be equivalent to the opening and closing bracket example from previously. This will parse any string, which has for one opening a exactly one opening b.

2.2 Lexer

The parsing process of a compiler is often divided into two parts. The first part is the lexer. The task of the lexer is to take the whole source code and create a token stream. The lexer works with regular expressions and avoids more complex implementations.

The lexer is a preparation for the parser, since it usually removes unwanted characters like whitespace and generalizes other tokens. For example, any string matching the regular expression `[a-zA-Z_][a-zA-Z_0-9]*` could be an identifier, and `[0-9]+` would be an integer. The name identifier is often used for variable, function and class names in lexers. This way the parser only has to work with the abstract identifier, not with the real input.

The following example illustrates the purpose of the lexer process

```
int get_fixed_sum(int a) {
    return a + 19284;
}
```

Listing 4: "Example input for the lexer"

The lexer will start parsing at the beginning, identify the first token, and continue. The resulting token stream might look like this.

```
int_type
identifier
bracket_open
int_type
```

```
identifier
bracket_close
curly_bracket_open
return_stmt
identifier
add
number
semicolon
curly_bracket_close
```

Listing 5: "Example output of the lexer"

Each line is a token. These tokens are used as terminal symbols in the grammar definition.

2.3 Parser

The second part will perform the parsing on the token stream and by using the grammar rules. There are many types of parsers, two of the common ones used in compiler development are

1. LL(1): Scanning the input from left to right, applying leftmost deriviation, using one token of lookahead.
2. LR(1): Scanning the input from left to right, applying rightmost deriviation, using one token of lookahead.

There are two goals of the parsing process. The first is to verify that the input conforms to the grammar. The second is to generate a tree structure that represents the input. The tree structure generated by the parser is called a Concrete Syntax Tree (CST). This tree structure is very verbose and represents the grammar definition exactly.

There are two main ways to parse source code. The first is top-down parsing. In top-down parsing, the root node is created first, followed by the child nodes, and the leaf nodes are created at the end. LL(1) parsing is an example of top-down parsing. The other approach is called bottom-down parsing. In this parsing strategy, the leaf nodes are generated first, and the root node is generated at the very end. LR(1) is an example of a bottom-down parsing algorithm.

LL(1) has the advantage of being relatively easy to understand, but LL(1) is less powerful than LR(1). LR(1) parsers are harder to understand and debug, but can parse more grammars then LL(1).

Each of the presented grammars are not able to parse all context free grammars, but they are sufficient enough to parse most programming languages. LR(1) is a parsing

strategy that is able to parse many grammars, but requires a large amount of memory and computation to parse correctly.

Since memory and runtime are not the main priority in this thesis, a LR(1) parser was implemented to allow a large number of grammars to be parsed.

The LR(1) parsing strategy relies on two different tables. The construction of the tables is explained in [ALSU06].

The following example is taken from [ALSU06].

Let $N = \{S, C\}, \Sigma = \{c, d\}, S = S$ a grammar with the productions P defined as

$$P = \{S \to C\ C \tag{1}$$
$$C \to c\ C \tag{2}$$
$$C \to d\} \tag{3}$$

Let r1 be a reference to the production rule $S \to C\ C$, r2 be a reference to $C \to c\ C$ and r3 be a reference to $C \to d$.

The first table is called the action table.

State	c	d	\$
0	s3	s4	
1			acc
2	s6	s7	
3	s3	s4	
4	r3	r3	
5			r1
6	s6	s7	
7			r3
8	r2	r2	
9			r2

The rows represent different states. The columns represent the terminal symbols of the grammar. The value of each cell can be empty, a shift, a reduce or an accept action.

A shift action will consume a token from the token stream and move to a new state. For example, s6 will perform a shift and move to state 6.

A reduce action returns to a previously encountered state and references a grammar rule that is being reduced.

An accept action is defined only once in the entire table. Once it is encountered, the parsing process is complete and the input has been successfully parsed.

The second table is the goto table.

State	S	C
0	1	2
1		
2		5
3		8
4		
5		
6		9
7		
8		
9		

The columns represent the non-terminal symbols of the grammar. The content of each cell can be either a reference to another state or empty.

The parser itself maintains a stack of states. The top of the stack is the current state being processed. The parser reads the current token from the token stream provided by the lexer and gets the current action from the action table. Depending on the action type, different behaviors apply.

On a shift action the parser will shift the position in the token stream and continue in the next iteration with a new token. The new state defined in the shift action is pushed to the stack. Then the next iteration is started.

On a reduce action, the parser will pop n states from the stack, where n is equal to the number of terminal and nonterminal symbols to the right of the grammar rule. Then the new top of stack is read and the state defined in the goto table, and the current token in the token stream is pushed onto the stack.

On an accept action, the parser will complete the parsing process by accepting the input.

3 Implementation

The program is implemented as a Java library and can be found in this github repository: `https://github.com/midob101/gp-modifiable-ast`.

In this chapter, a `user` is a software developer who uses this library.

3.1 Architecture

The only dependencies used are JUnit and JUnit-jupiter [jun] for testing purposes.

The following figure illustrates the architecture and design of the `gp-modifiable-ast` library. Each small rectangle represents an entity of the entire program, the small rectangles with **bold** text are user-defined inputs and implementations and the ones with *italic* text are implementations of the `gp-modifiable-ast` library.

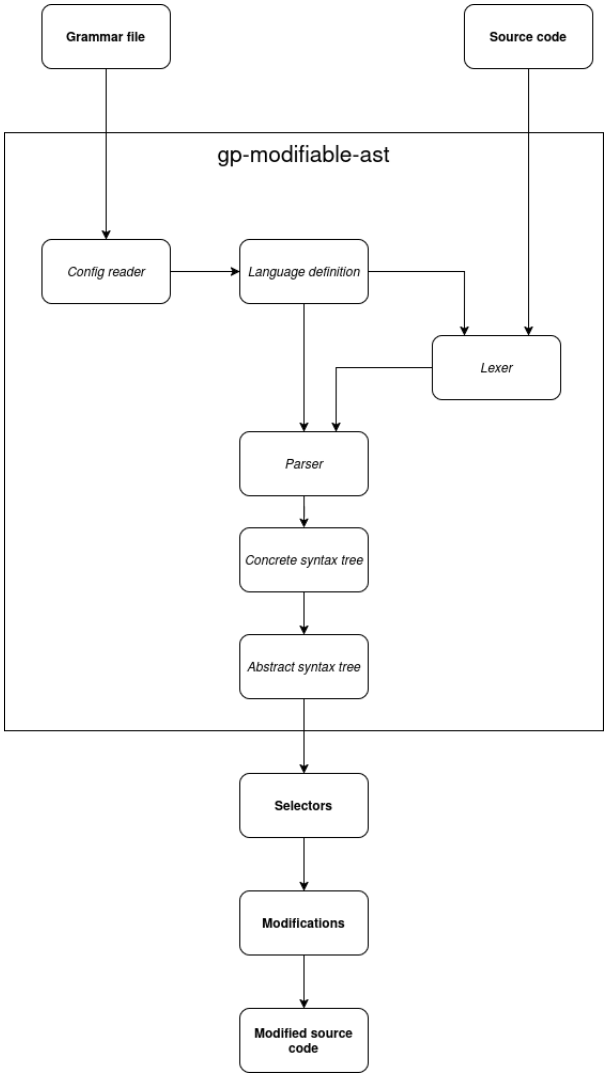

Figure 2: Architecture of gp-modifiable-ast

The user defines a grammar file for a given programming language and provides the source code to `gp-modifiable-ast`.

`gp-modifiable-ast` first parses the grammar file and creates a language definition that contains the grammar productions, lexer rules, and generic settings for the language.

This language definition is then passed along with the source code by the user to the lexer process, which passes the token stream and the language definition to the to the parser process. The parser process parses the token stream, applies the grammar rules and creates the concrete syntax tree.

In the next step, the concrete syntax tree and the grammar definition will be used to create the abstract syntax tree.

This is the output of the `gp-modifiable-ast` library. By using selectors, which will be defined later, the user can perform search actions on the AST to find specific nodes. On the nodes found, the user can apply modifications and convert the AST back to source code.

3.2 Class generation vs general classes

Most parser generators will generate source code for the parser. This allows the user to modify the generated classes, for example by adding new methods. This allows better usability if done correctly, and better performance. Instead of generating code for the parser, gp-modifiable-ast will use generic classes that are the same for all grammars being parsed. This approach does not require any additional code generation by the parser and allows an easier introduction to the system, since a language definition and a source file can simply be passed to the parser generator and an AST is generated. Otherwise, the parser must first be generated and then the parser must be called to parse the source file.

3.3 Grammar file syntax

The productions and lexer definitions are defined separately from the implementations in their own text files. Each language has exactly one text file defining the productions and lexer definitions. The contents are divided into four different chapters.

The following example shows a grammar for arithmetic expressions that could be found in a programming language grammar.

This grammar allows to parse simple arithmetic expressions and allows comments and whitespaces.

```
1 LANGUAGE_DEF
2     grammar_start = S;
3
4 LEXER_RULES
```

```
    add = "+";
    subtract = "-";
    multiply = "*";
    divide = "/";
    bracket_open = "(";
    bracket_close = ")";
    integer = regex(\d+);
    whitespace = regex(\s+);
    single_line_comment = customMatcher(singleLineCommentMatcher);
    multi_line_comment = customMatcher(multiLineCommentMatcher);

HIDDEN_LEXER_RULES
    whitespace, single_line_comment, multi_line_comment;

PRODUCTIONS
    S   ->  S[alias=left] add T[alias=right] |
            S[alias=left] subtract T[alias=right] |
            T;
    T   ->  T[alias=left] multiply F[alias=right] |
            T[alias=left] divide F[alias=right] |
            F;
    F   ->  bracket_open S bracket_close |
            integer;
```

Listing 6: Grammar file example

This grammar file would be able to parse the following sample code:

```
// Arithmetic grammar test
(5+10) /** inline comment */
* (15*20)
```

Listing 7: Grammar file example

The first chapter is `LANGUAGE_DEF`. This contains some general definitions for the language. This chapter contains e.g. which comment styles are allowed and how they are identified, the name of the language, the file extension used, whether the language is case-sensitive, and which production should be used as the starting production, whether the language is case-sensitive or not, and which production should be used as the starting production. All settings here are defined as key-value pairs.

The second chapter is the `LEXER_RULES`. This defines all the tokens that can occur in the source file and that the lexer should handle. The lexer can handle three different types of definitions. The first type matches a fixed string. The second type matches a regex. The last type uses a custom implementation to check for a match. The last matching definition wins. For example, the token definition `true_literal = "true";` and the definition `identifier = regex([a-zA-Z_][a-zA-Z_0-9]*);` would both match the string `true`. Therefore, the more general `identifier` token should be defined before the `true_literal` token. This will cause the lexer to recognize the text `true` as `true_literal`. A lexer key must be written in all lowercase letters and underscores.

The chapter HIDDEN_LEXER_RULES is a comma separated list of lexer definition keys, that should not be handled in the parsing process and should not be visible in the AST by default. This category should be used mainly for whitespace and comments. If these are not added to the category, the grammar rules must handle them. Since whitespace and comments can be added basically anywhere, this would result in a very verbose definition of productions.

The chapter GRAMMAR_RULES contains all productions of this language. The syntax is similar, but not identical, to the backus natus form. (TODO: Source). The syntax has been extended with most of the modifiers suggested by [OJ09]. The terminals are the keys of the lexer definitions. , The nonterminals are defined by the left sides of the productions. All nonterminals must be written in uppercase and underscore. EPSILON is used for the empty word. Each terminal and nonterminal symbol can be extended by a list of modifiers in square brackets after the name.

`S -> S[alias=left] add T[alias=right]` defines the production with `S` on the left and `S add T` on the right. `S` refers to the same production, `T` refers to a different production. `add` refers to a lexer definition. `[alias=right]` sets an alias in the AST for the `T` production.

3.4 CST-Generation

The concrete syntax tree (CST) is a tree structure that exactly matches the grammar and lexer definitions. The generated CST should also contain nodes that are usually omitted, such as comments or whitespace.

The CST is implemented by a `ConcreteSyntaxTreeNode` class. A node can refer to either a production or a lexer token. We keep the productions because they allow us to make advanced modifications and generate a less verbose abstract syntax tree from the CST. We keep the lexer tokens because they contain the original sources.

This structure is generated directly by the parser. The parser maintains a second stack containing tree nodes that are not bound to a parent node.

As the parser passes through the token stream provided by the lexer, it generates new tree nodes and pushes them onto the stack whenever either the token is in HIDDEN_LEXER_RULES or the action performed is a shift action.

On a reduce action the parser will also create a new tree node. This tree node will have other tree nodes from the top of the stack as children. This behavior is very similar to how the the parser manages its state. The main difference is the tokens in the HIDDEN_LEXER_RULES. The reduced production may have only one symbol on the right side, but on the tree node stack there are two nodes from HIDDEN_LEXER_RULES. In this case the parser will pop three states from the top of the tree node stack and add them as

children of the newly created node in reverse order. However, if the parser reaches the `acc` state, it is not guaranteed that the stack contains only the root node. There may be tokens from `HIDDEN_LEXER_RULES` which appeared at the beginning of the parsed string. So the parser will find the root node and prepend all other entries in the tree node stack to the children of the root node.

With this algorithm, every single token from the lexer's token list is transformed into a tree node and placed in the CST. This guarantees that the whole source code is represented in the CST and that the CST could be transformed back to the source code exactly.

The following code shows a simplified version of the CST generation during the parser process.

```
if(isHiddenLexerRule() || isShiftAction()) {
    danglingTreeNodeStack.push(new TreeNode(currentParseAction));
}

if(isReduceAction()) {
    TreeNode newTopOfStack = new TreeNode(currentParseAction);
    int i = getRightHandSymbolCount();
    while(i > 0) {
        TreeNode currentTopOfStack = danglingTreeNodeStack.pop();
        newTopOfStack.pushChild(currentTopOfStack);
        if(!currentTopOfStack.isHiddenLexerRule()) {
            i--;
        }
    }
    danglingTreeNodeStack.push(newTopOfStack);
}

if(isAcceptAction()) {
    TreeNode rootNode = danglingTreeNodeStack.pop();

    while(!danglingTreeNodeStack.isEmpty()) {
        TreeNode treeNode = danglingTreeNodeStack.pop();
        rootNode.prependChild(treeNode);
    }

    return rootNode;
}
```

Listing 8: CST Generation

3.5 AST-Generation

Even though the CST already contains all relevant information about the sources, it is very verbose, and the structure of the CST is usually not very friendly to work with. For this reason, `gp-modifiable-ast` implements the modifiers provided by [OJ09], which are used to convert the CST into an AST.

There are three types of nodes in the AST. `ProductionTreeNode` are nodes derived from a grammar production. `TokenTreeNode` are nodes derived from a lexer token. `StringTreeNode` are nodes that contain a string and are used to replace other tree nodes. The `StringTreeNode` is not part of the initial AST. This node can be used by the user to replace subtrees in the final AST or to add a new node.

Any `TokenTreeNode` that references a token from `HIDDEN_LEXER_RULES` is marked as hidden. These nodes are stored in the AST, but are not visible to the user unless specifically requested. Next, the AST structure is modified by the modifiers defined in the grammar file. Each modifier has its own implementation and is applied bottom-up for each node.

3.5.1 list-Modifier

The `list` modifier, intended by [OJ09], is a modifier designed to flatten out self-recursive productions. This modifier can only be applied to symbols on the left side of a production. The following example could be a production for the function parameters in common programming languages.

```
1 PARAMETER_LIST[list] -> PARAMETER_LIST PARAMETER |
2                         PARAMETER |
3                         EPSILON;
```

Listing 9: list modifier example

This grammar rule generates the following CST for an input with three parameters.

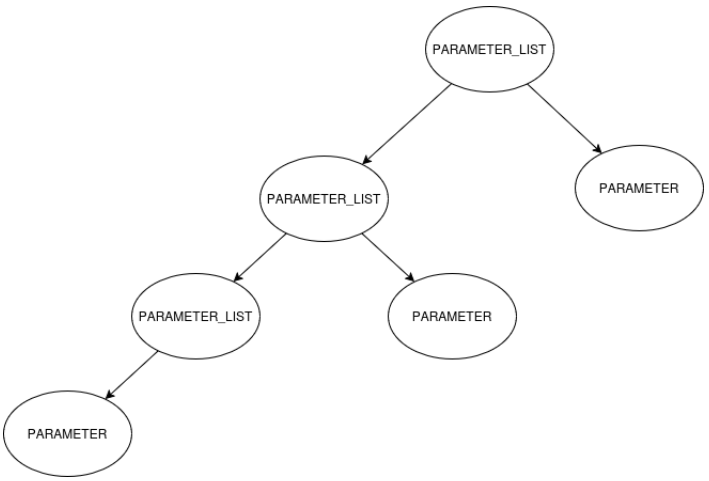

Figure 3: CST before the list modifier is applied

This structure is too verbose and difficult to work with in practice, so by using the `[list]` modifier, we get the following AST.

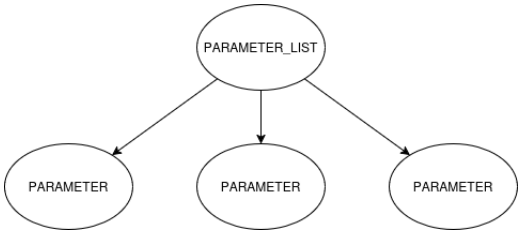

Figure 4: AST after the list modifier is applied

This structure is much easier to manage and still contains all relevant information about the sources.

The modifier is applied by checking if the children of the current `ProductionTreeNode` n contain a `ProductionTreeNode` m that references the same production. If so, m is replaced in the children list of n by all children of m. Since the modifiers are applied bottom-up, this works for nested trees.

3.5.2 alias-Modifier

The `alias` modifier, purposed by [OJ09], is a modifier that can be applied to any symbol on the right hand side of a production. This modifier, adds an alias to the tree node that can be used to search the tree. The following could be an example of a production for addition in a programming language.

```
ADD -> NUMBER[alias=left] plus NUMBER[alias=right];
```

Without the alias modifier, the only way to distinguish between the two `NUMBER` nodes would be by the order of the children of the `ADD` tree node. The `alias` modifier allows for cleaner searches in the AST.

This modifier is applied by simply storing the alias in the tree node.

3.5.3 hidden-Modifier

The `hidden` modifier, purposed by [OJ09], is a modifier that can be applied to terminal symbols on the right hand side of a production. This modifier will hide the tree node in the AST. In the previous example of the `ADD` production, the `ADD` tree node would have a `TokenTreeNode` child that references the `plus` lexer definition. This information is obsolete because the production will always contain this node, and the production name already

contains the necessary information. By applying the `hidden` modifier to the `plus` symbol, the corresponding tree node will still exist, but will not be visible unless specifically requested.

3.5.4 inline-Modifier

The `inline` modifier, purposed by [OJ09], is a modifier that can be applied to nonterminal symbols on the right hand side of a production.

Applying the modifier will replace the node with all its children. This modifier should be used on nonterminals that do not carry important information themselves, but their children do. This way all information is preserved, but the tree structure is simplified.

3.5.5 Not implemented modifiers

[OJ09] purposed additional modifiers that are not implemented in `gp-modifiable-ast`. The first is the `Boolean Access` modifier, which would replace a node with a boolean value depending on whether that symbol was parsed. However, since we do not generate classes for every production, this would be of little use to us. The same reason applies to the `superclass` modifier, which would create a hierarchy in the generated classes of the parser generator.

3.6 AST search methods

Before any changes can be made to the AST, the nodes that need to be changed must first be found.

For this approach, each AST node has three methods.

1. `query: Selector -> QueryResult`
2. `queryChildren: Selector -> QueryResult`
3. `queryImmediateChildren: Selector -> QueryResult`

The `query` method takes a selector and returns a `QueryResult`. The `QueryResult` contains all nodes in the subtree of the searched node that match the `selector`. The `QueryResult` may also contain the searched node itself if it matches the `selector`.

The `queryChildren` behaves the same way, but does not include the searched node itself.

The `queryImmediateChildren` will return only the immediate children of the searched node that match the `selector`.

The `QueryResult` instance also allows to perform queries on the result. This will perform the corresponding method on all nodes in the result and create a new `QueryResult` instance containing the merged results of each node. This allows for easy chaining of selectors. You can also merge `QueryResult` instances for further processing with the `merge` function.

3.7 Selectors

A `Selector` is a class instance that should test whether a given tree node matches the rules defined by the `Selector`. The `Selectors` can be divided into three different categories.

The first category are logical `Selectors`. They should represent logical operations. There are two `Selectors` defined by default, `AndSelector` and `OrSelector`. Each takes a list of other selectors. The `AndSelector` will only match tree nodes that match all passed `Selectors`. The `OrSelector` will match all tree nodes where at least one of the passed `Selectors` matches the node.

The second category are data `Selectors`. These selectors search for nodes based on the data they contain. They can match the name of a production, the alias of a production, the name of a lexer definition, and the value of a lexer token. The implemented selectors are `AliasSelector`,`ProductionSelector`,`TokenSelector`,`TokenValueSelector`. All of these take a string as a parameter and check the corresponding notes for that string.

The last category are structural `Selectors`. These selectors search for nodes that have specific nodes in their children or parents.

The following example of an AST is intended to illustrate the purpose of this category.

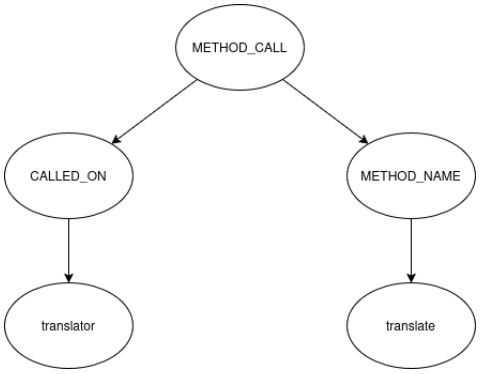

Figure 5: AST example for selectors

To search for all `METHOD_CALL` nodes, that call translate on a translator object, we will want these types of selectors.

An example selector for this would be

```
new AndSelector(
  new ProductionSelector("METHOD_CALL"),
  new HasImmediateChildSelector(
    new AndSelector(
      new ProductionSelector("CALLED_ON"),
      new HasImmediateChildSelector(
        new TokenValueSelector("translator")
      )
    )
  ),
  new HasImmediateChildSelector(
    new AndSelector(
      new ProductionSelector("METHOD_NAME"),
      new HasImmediateChildSelector(
        new TokenValueSelector("translate")
      )
    )
  )
)
```

Listing 10: Selector example

By calling the `query` method on the root node of the AST, we get all the `METHOD_CALL` nodes in the AST that match those conditions.

3.8 AST Modifications

The `AbstractSyntaxTreeNode` class contains several methods for making changes.

1. `replace` Replaces this node with one node or a list of other nodes.
2. `replaceChild` Replaces a node in its children with one node or a list of other nodes.
3. `remove` Removes this node and its subtree.
4. `removeChild` Removes a child.
5. `addChild` Adds a child to the end of the child list.
6. `addChildBefore` Adds a child before another node.
7. `addChildAfter` Adds a child after another node.
8. `deepClone` Clones this node and all its children, returning a node that is the same as the current one, but without a parent reference.

Each `AbstractSyntaxTreeNode` references its corresponding parent node. To maintain integrity, you can only add nodes that have no parent node, and only replace nodes with a parent reference. To add a node to several other nodes, you can clone the node to be added.

3.9 Extendability

To allow for possible extensibility, we allow for several types of customization

3.9.1 Lexer

As mentioned before, lexer definitions can contain custom matchers. These matchers are custom implementations that are referenced by a name in the grammar file. These classes are written by the user and registered by calling `CustomMatcherRegistry.registerCustomMatch`
A custom matcher must implement the `ICustomMatcher` interface. A matcher gets a `LexerContext` and returns `null` or the matched string. Using the `LexerContext` parameter, the matcher can receive the remaining input and check if the beginning of the remaining input can be tokenized by this matcher.

The following example is an implementation of a custom matcher, that would match single line comments.

```
1  public class ExampleCustomMatcher implements ICustomMatcher {
2      @Override
3      public String match(LexerContext context) {
4          if(context.getNextNChars(2).equals("//")) {
5              // Find the end of line. This will be the comment token
6              String remaining = context.getRemainingSource();
7              Pattern p = Pattern.compile(".*(\r\n|\r|\n|$)");
8              Matcher matcher = p.matcher(remaining);
9              if (matcher.find()) {
10                 return matcher.group();
11             }
12         }
13
14         return null;
15     }
16 }
```

Listing 11: Example of a custom matcher

3.9.2 AST Generation

We allow several ways to customize AST generation. First, you can implement custom modifiers for grammar symbols. You can also post-process and decorate the AST. For example, you can replace the tree nodes with your own implementation by extending `AbstractSyntaxTreeNode`.

The following example is an implementation of a tree node that is simply a string. This tree node can be used to add or replace code in the AST.

```
public class StringTreeNode extends AbstractSyntaxTreeNode {
    private final String value;

    public StringTreeNode(String value) {
        this.value = value;
    }

    public String getDisplayValue() {
        return "StringTreeNode, value: " + this.value;
    }

    public String getValue() {
        return value;
    }

    protected String getSources() {
        return this.value;
    }
}
```

Listing 12: Implementation of StringTreeNode

3.9.3 Selectors

Custom selectors can be defined by implementing `ISelector`. These can be easily defined and used with the query methods of the tree node.

The following example shows the implementation of a selector that matches a tree node based on the production used to create that node.

```
public class ProductionSelector extends BaseSelector {
    private final String production;

    public ProductionSelector(String production) {
        this.production = production;
    }

    @Override
    public boolean matches(AbstractSyntaxTreeNode treeNode) {
        if(treeNode instanceof ProductionTreeNode convertedNode) {
```

```
11              return convertedNode.getProduction()
12                                  .leftHandSymbol()
13                                  .name()
14                                  .equals(this.production);
15          }
16          return false;
17      }
18  }
```

Listing 13: Custom selector implementation

4 Implemented Grammars

There are two grammars implemented in `gp-modifiable-ast` by default. The first grammar is MiniJava. The MiniJava language was defined by [App02].

The second grammar is an extension to MiniJava, which was defined for this thesis. This extended grammar allows for more operations in comparisons. This grammar is only used to illustrate some possible refactorings and use cases of `gp-modifiable-ast`.

4.1 MiniJava

The implemented grammar is based on the transformed grammar in Backus Natus Form from [min].

The MiniJava grammar has been defined by `gp-modifiable-ast`. This is a very limited subset of Java, but it represents common constructs found in modern programming languages. It includes basic object oriented programming, simple conditional statements, while loops and basic arithmetic operations. However, the grammar is quite limited. For example, it does not allow any other comparison operator then < and not more then one arithmetic operation without brackets. As these limitations make it hard, to create meaningfull refactorings, we have extended the syntax of MiniJava aswell to allow for other operations.

4.2 Extended MiniJava

The extended MiniJava grammar includes a small change to the grammar.

The grammar allows the use of more comparison operators. MiniJava only allows <.

Extended MiniJava includes the other comparison operators >, >=, <= and ==.

Each of those comparison operators can be refactored to <, as MiniJava only supports integers and not floating point values.

The purpose of extended MiniJava is to perform a source code transformation, which converts any source code written in extended MiniJava to the the MiniJava syntax. These types of transformations are commonly used in practice, for example by Babel [bab], which is used to transform modern JavaScript into old JavaScript syntax. This allows developers to write code using the latest JavaScript standard while still supporting old browsers, that run transformed code.

5 Use cases

In this chapter we would like to show some refactorings that have been implemented on MiniJava and some real refactoring examples, but that have not been implemented.

5.1 Yoda conditions

Yoda conditions are conditions in the form `a compop b`, where `a` is a literal (e.g. 5) and `b` is a expression. Non Yoda conditions are conditions in the form `b compop a`, where `a` is a literal and `b` is a expression.

Usually in software development, you want to follow one style and use it for the whole project. Therefore, various IDEs and other tools like lexers exist, which check the styling and if possible fix them automatically. These tools may not exist for domain specific languages, therefore a usecase of this project can be those refactorings.

To implement this refactoring, we are using the extended MiniJava syntax. At first, we are applying a selector to find all comparison operators in the AST. From the result, we can simply grab the left and right side of the node, as they are decorated with an alias. If the left side is an integer literal, we swap the left and right node. At the end, we need to swap the `compop` to the reflect the changes. < will become >=, > will become <=, <= will become >, >= will become <. == will not need to be adjusted.

As we chose to add the whitespaces as real nodes to the AST, even though they are hidden, we do not mess up with the whitespaces. The node that corresponds to `a` does only contain the `a` constant. Any whitespaces between `a` and `compop` will be a seperate node in the AST. Therefore, by swapping a and b the formatting of the file stays the same.

The following code is a snippet from the actual refactoring implemented in [Dobb]

```
1 if(leftIsLiteral && !rightIsLiteral) {
2     left.replace(right.deepClone());
3     right.replace(left.deepClone());
```

```
4
5       String newCompOp = compop.getValue();
6       switch (compop.getValue()) {
7           case "<":
8               newCompOp = ">";
9               break;
10          case "<=":
11              newCompOp = ">=";
12              break;
13          case ">":
14              newCompOp = "<";
15              break;
16          case ">=":
17              newCompOp = "<=";
18              break;
19      }
20      compop.replace(new StringTreeNode(newCompOp));
21  }
```

Listing 14: Refactor Yoda conditions

5.2 Translation system

This is an example for a refactoring, which may be required to be performed on a large scale and that is not doable by common tools.

Let a translation system be a system, which stores for a key some data. This data includes the translations for each desired language and parameters. A parameter is a variable, which gets dynamically inserted into the text. One of the parameters can be used as a pluralization parameter.

For example, the key `"apple"` resolves to the english translation `"{NAME} has {COUNT} apple"` if the parameter `COUNT` is 1 and `"{NAME} has {COUNT} apples"` if `COUNT` is not 1. `{NAME}` is a second parameter, which does not require any pluralization.

The wanted translation is received by calling `translate("apple", {name: user, COUNT: i})`.

When switching to another translation system, which requires the pluralization parameter to be passed seperately, we are running into an issue. The desired call syntax will be: `translate("apple", {name: user}, {COUNT: i})`

The old code does not include any information, which of the parameters is the pluralization parameter. We require the connection to an external source to correctly identify the wanted parameter.

There are many difficulties to perform this refactoring, and no common IDE tools can help us with this. We might be able to perform this action with regex replacements, but we would need to write or generate a seperate regex for every single translation. Even if

we get each different regex, there might still be more issues, like complex definitions of the parameters and it will be not possible to parse every possibility of the call syntax, as regular expressions are not able to parse context free grammars which are not regular. In the worst case, unrelated code that should not have been modified gets changed and can lead to additional issues.

There are different options to handle this case. We could do every refactoring manually, but this will be very time consuming if there are many translations. We could also implement the adapter pattern, which would allow us to hide the new API from the old code. Or as a last option, we could implement a custom solution.

With the gp-modifiable-ast-parser we are able to parse the sources, receive an AST and identify the translation key and all parameters passed. To identify which parameter is the pluralization parameter, we can receive the definition of the translation. Afterwards, we can simply remove the pluralization parameter from the call syntax and add it as a new parameter to the function call.

5.3 Code transpilation

This project may also be used to transpile code from a source grammar to a target grammar. As an example, a transpiler was written, which transpiles applications that use the extended MiniJava grammar to applications that use the regular MiniJava grammar. An example of a transpiler which performs operations like these would be babel. This tool is used to transpile a newer version of JavaScript into an older one, which can be executed by more browser. That allows the developer to utilize new functionalities of the programming language while not breaking backwards compability.

The goal of this implementation is to show the power of a rewritable abstract syntax tree. Especially for extended return syntax, we have to apply severe modifications to the sources to make it MiniJava compatible.

5.3.1 Conditionals

As MiniJava only supports the less then operator, we need to transform every conditional that uses a different operator. For this, we use the following equivalent statements for $a, b \in \mathbb{Z}$:

$$a > b \iff b < a \tag{4}$$
$$a \leq b \iff a < b + 1 \tag{5}$$
$$a \geq b \iff b \leq a \iff b < a + 1 \tag{6}$$
$$a = b \iff (a \leq b) \land (a \geq b) \iff (a < b + 1) \land (b < a + 1) \tag{7}$$

As we have reduced all new comparison operators to the less then operator now, we can implement the refactoring.

This is done by creating a selector to find all comparisons, reading the comparison operator and performing the refactoring. The following code is an implementation of the transpilation process. The complete source is accessible at [Doba].

```
AbstractSyntaxTreeNode ast = gpModifiableAST.createAst(input);
QueryResult comparisons = getComparisons();
for(AbstractSyntaxTreeNode comparison: comparisons) {
    AbstractSyntaxTreeNode left = getLeftNode();
    TokenTreeNode compop = getCompOpNode();
    AbstractSyntaxTreeNode right = getRightNode();

    switch (compop.getValue()) {
        case "<=":
            compop.replace(new StringTreeNode("<"));
            right.replace(List.of(
                    new StringTreeNode("("),
                    // As right is still attached to the tree, we need to
                    // clone the node.
                    right.deepClone(),
                    new StringTreeNode(" + 1)")
            ));
            break;
        case ">":
            compop.replace(new StringTreeNode("<"));
            left.replace(right.deepClone());
            right.replace(left.deepClone());
            break;
        case ">=":
            compop.replace(new StringTreeNode("<"));
            left.replace(right.deepClone());
            right.replace(List.of(
                    new StringTreeNode("("),
                    left.deepClone(),
                    new StringTreeNode(" + 1)")
            ));
            break;
        case "==":
            // Replace equality operator
            break;
    }
```

```
37 }
```

Listing 15: Code transpilation

5.4 Code analysis

The gp-modifiable-ast can be used to perform code analysis. Even though this was not the main intention, the requirements of the project is equal to the ones required for code analysis.

All comments and whitespaces are kept. Specific comments are often used to disable code analysis for certain parts of code. Whitespaces are required to check code formatting.

However as the gp-modifiable-ast parser is not implemented for a specific programming language, some features cannot be implemented in general. For example a symbol table cannot be implemented for all programming languages in the same way.

The main benefit of using gp-modifiable-ast for code analysis would be that some of the analysis actions could be defined ones and used for multiple programming languages.

It would still be required to pass configurations for each programming language into the analysis action.

6 Conclusion and further work

The implemented library is capable of generating an AST that preserves all information about the original sources. This is done by parsing a language definition file, building a LR(1) parser, parsing the source code, and building an AST which still contains all the tokens that are usually ignored by parsers. With the provided API, it is possible to traverse and modify the AST in a way to implement custom modifications to a given source.

The grammar for MiniJava has been defined and several refactorings have been implemented to show the functionality and usability of the implementation.

A limiting factor is the effort required to define a language grammar. This could be improved by extending the grammar's syntax to accept the extended backus natus form.

Another problem, due to the fact that the library is not implemented for a specific programming language, is the lack of functionality of complete compiler frontends, including but not limited to symbol tables. If there is a way to to create symbol tables and track the usage of certain variables, this library could be used for more refactorings. Currently, these functions must be built on top of the library.

Therefore, this library is currently best used for refactorings to single nodes of an AST that contain all the information needed to perform the refactoring.

List of Figures

List of Tables

References

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[ant] Antlr. `https://www.antlr.org/`. Accessed: 2024-06-08.

[App02] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002.

[bab] Babel. `https://babeljs.io/`. Accessed: 2024-12-29.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Doba] Michael Doberstein. Extended minijava transformation. `https://github.com/midob101/gp-modifiable-ast-examples/blob/main/src/main/java/org/example/TransformToMiniJava.java`. Accessed: 2024-12-29.

[Dobb] Michael Doberstein. Yoda refactoring. `https://github.com/midob101/gp-modifiable-ast-examples/blob/main/src/main/java/org/example/RefactorYoda.java`. Accessed: 2024-12-29.

[esp] esprima. `https://esprima.org/`.

[gnu] gnu-bison. `https://www.gnu.org/software/bison/`.

[jun] Junit 5. `https://junit.org/junit5/`.

[Lin02] Peter Linz. *An Introduction to Formal Languages and Automata*. Cambridge University Press, 2002.

[min] Minijava bnf. `https://www.cse.iitm.ac.in/~krishna/courses/2014/even-cs6848/minijava.html`. Accessed: 2024-06-08.

[OJ09] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, pages 114–133, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[sab] sablecc. `https://sablecc.org/`.